PSynergy

An Independent Journal for PlayStation 2 Linux Developers and Enthusiasts

Issue 3, April 27, 2004.

Table of Contents

EDITORIAL **2**

THIS MONTH IN THE FORUMS **3**

FEATURES **5**

FEATURE: USING THE DMAC IN GAMES PROGRAMMING 5
Using Call Chains in Games Programming *9*
Figure 5 ..... *10*
Conclusions *11*
Acknowledgements *11*
Dr Henry S Fortuna *11*
FEATURE: MSKPATH3 TUTORIAL AND COMMENT 12

INTERVIEW **17**

PS2 DEVELOPMENT TIPS **19**

NEXT MONTH IN PSYNERGY **20**

End-Reader License

You my freely reproduce and redistribute this publication **in its entirety** through any means.

You may freely redistribute any article contained herein **in its entirety** as long as the name of the original author and the text "as per The PSynergy Journal" are placed both at the beginning and the end of the article. Also, this license paragraph must be copied and appended to the article to ensure that it continues to apply to the redistributed article.

Article Submission Guidelines

You may submit a copy of your article manuscript to patryk@pakl.net. For your article to even be considered, it must be related to development for the Playstation 2 Linux kit. Please note that this includes programming for the RTE, so long as anyone with the Kit can learn from/use it. All submissions will be edited for content, clarity and brevity and the author is given at least 3 days to approve final copy. A suggested template for articles will be provided at http://psynergy.pakl.net/ template.html. Finally, by submitting your article, you agree that it can be redistributed freely, only in its entirety, as long as your name is placed at the beginning and the end of it.

"Sony" is a registered trademark of Sony, Inc. "PlayStation", "PlayStation2", the PlayStation "PS" logo, and all associated logos, are trademarks of Sony Computer Entertainment, Inc. "PSynergy" and is NOT associated in ANY way with Sony Computer Entertainment, Inc.

Editorial

Readers,

Here finally is Issue 3. The combination of tax day and final exams made this one particularly late.

Happy Spring/Summer!

Patryk Laurent
Editor-in-Chief, PSynergy
patryk@playstation2-linux.com

This Month in the Forums

A Monthly Column by Eratosthenes

Apologies for my recent total absence from the PS2 scene - I've been a busy guy! However, there's nothing like a good browse through the Developer's forum to catch up on the latest...

VU C Compiler

In what sounds to me like the most significant step in PS2Linux tools since the creation of SPS2, we now have a C compiler for the VUs! Yes, that's right, create VU code in C! This deserves an entire article of its own - hopefully you'll find one elsewhere in this very issue of PSynergy!

Sauce's MSKPATH3 Pseudo-Tutorial

With any luck you'll also find Sauce's Pseudo-Tutorial on the MSKPATH3 technique in this issue. This technique is used to manage your textures frame by frame, and is one of those things you need to read about a few times before the magic is somewhat understandable! Sauce did a great job in the forums, go check out this article for another read :)

MinRay for SPS2

At http://playstation2-linux.com/forum/message.php?msg_id=42061 lives an ideal candidate for a code snippet if I ever saw one, a Minimal Ray Tracer for SPS2. Print it out in a small enough font and the source would probably fit on a business card...

MIPS1 Compilation Issues

A discussion at https://playstation2-linux.com/forum/message.php?msg_id=41740 shows some problems that vliw hit when he tried using inline MIPS asm. Sparky verified that the code works on his T10k at work, so the problem was narrowed down to GCC and the MIPS1 barrier. Any potential compiler hackers should read this thread and maybe write an article about the MIPs1 limitations ;)

Higher level library wanted

The discussion at https://playstation2-linux.com/forum/message.php?msg_id=42332 centers around the creation of a higher level library for programming the PS2. The general consensus seems to be that a general higher level library is not the way to go to

get maximum performance, but what do you think? Can a higher level library reach an acceptable performance level for people who don't want to mess with ASM and VU coding? Do have the desire to write such a library? Would you use it if it were created? **Answers on a postcard to the PSynergy mailbox, perhaps there's an article in this for a future issue...**

Other news in brief

* Win32/Cygwin cross-compiler in CFYC: If you want to compile on Cygwin whilst your PS2 is playing games, you need this!

https://playstation2-linux.com/files/cfyc/gcc-2.95.2-ps2linux-win32.zip

* Developers Wanted for a hardware benchmarking suite for the PS2.

http://playstation2-linux.com/projects/psb/

* Kazan (Jonathan Hobson) has released some excellent SPS2 demos with source. (Yay! Metaballs!)

http://www.jhobson.co.uk/ps2section.htm

Features

Feature: Using the DMAC in Games Programming

Dr Henry S Fortuna (h.s.fortuna@abertay.ac.uk)
University of Abertay Dundee, Scotland UK

Introduction

Background information on the general operation and characteristics of the Direct Memory Access Controller (DMAC) was provided in the article "The PS2 Direct Memory Access Controller" published in PSynergy Issue 2 – March 2004. This article will describe how the various Direct Memory Access Control Tags (DMATags) can be used to help manage the transfer of model and texture data through the graphics pipeline of the PlayStation2 in a typical Computer Game application.

Background

The internal structure and main data paths within the PS2 are shown in figure 1. The DMAC is responsible for transferring data between main memory and each of the independent processors and between main memory and scratchpad RAM.

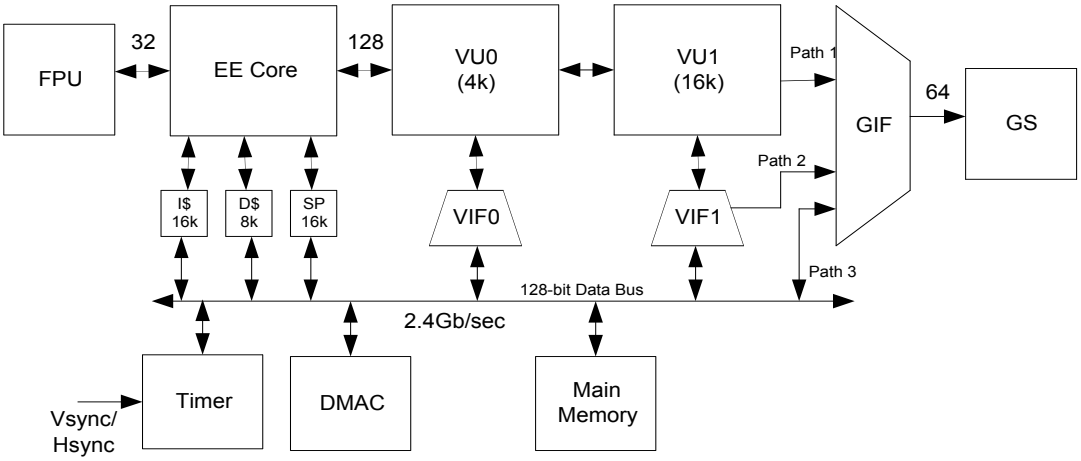


Figure 1

During the execution of typical game code, the DMAC is responsible for transferring vertex data and transformation/lighting matrices to Vector Unit 1 (VU1), and image data

for primitive texturing to the Graphics Synthesiser (GS). In order to maintain an effective frame rate it is important that as much of this data as possible is pre-compiled and efficiently organised prior to run time. Such organisation frees up the main processor from this mundane task and allows it to perform other important game related functions such as AI and game logic during game execution.

Image Data Transfer

Image data used for texturing is normally sent to the GS via path 2 or 3. Path 3 is a direct path to the GIF whilst Path 2 is through VIF1 to the GIF. There are a few additional overheads associated with sending data via Path 2 but Path 2 has the advantage of providing inherent synchronisation between texture and vertex data.

Typical image data may be many KiloBytes in size and generally larger than the 4 kByte memory block allocation size provided under SPS2. It is therefore necessary to split the image data into 4 kByte blocks and stitch these blocks together with appropriate DMATags. As discussed above, such organisation of the texture data should be undertaken prior to run time. Achieving this with memory stitching is outlined below.

Memory Stitching

The process of pre-compiling image data will be demonstrated using two different methods of memory stitching. The first method uses **cnt** and **next** tags and the second uses **ref** tags.

Organising data with **cnt** and **next** tags is illustrated in Figure 2. A **cnt** tag with it's qword count field (QWC) set to 254 is inserted at the start of each full 4k block. The value in the address field (ADDR) is not used with **cnt** tags and can be cleared to zero. The **cnt** tag instructs the DMAC to transfer QWC of data following the tag, and read the quad word after that data as the **next** DMATag, which in this case is a **next** tag. The purpose of the **next** tag is to direct the DMAC to the start of the **next** 4k block to be transferred. This is achieved by setting the ADDR field of the **next** tag to point to address A1 (which is the start of the next 4k block) and the QWC field of the tag to zero to indicate that no data is to be transferred with this tag. The DMAC therefore reads the **cnt** tag at address A1 as the **next** instruction and this process repeats until the last block is reached. The QWC of the **cnt** tag in the last block is set to the amount of data to be transferred and the transfer process is ended by inserting an appropriately configured end tag after the final data section.

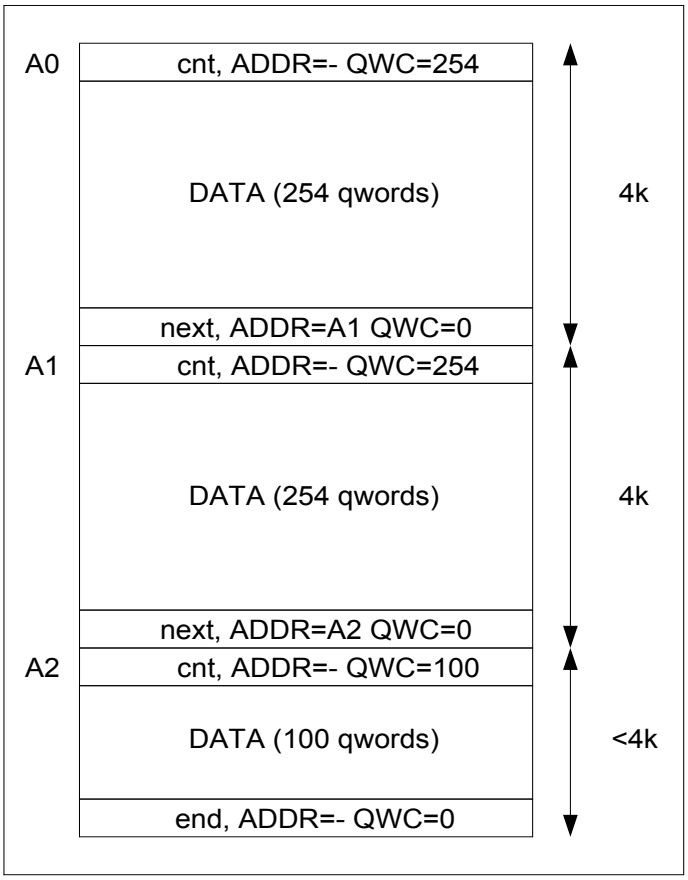


Figure 2

It is interesting to note that the final **end** tag could be replaced with a **ret** tag if the data packet is part of a call chain, but this will be described in more detail later in this article.

Organisation of data with **ref** tags is illustrated in figure 3. In this case, the 4k block contain only the data to be transferred and there are no embedded DMATags within the data. A separate area of memory is required to build the DMAC command chain which is constructed using **ref** tags and ended with a **refe** tag. The tag at address A3 is the first to be read and this instructs the DMAC to transfer the 4k block starting at address A0 then read the tag after the one at A3 as the next tag. This process continues until the final **refe** tag is reached, this transferring the final section of data then ending the transfer. In this case, if the DMA chain is part of a call chain the final **refe** tag can be replaced by an appropriately configured **ret** tag.

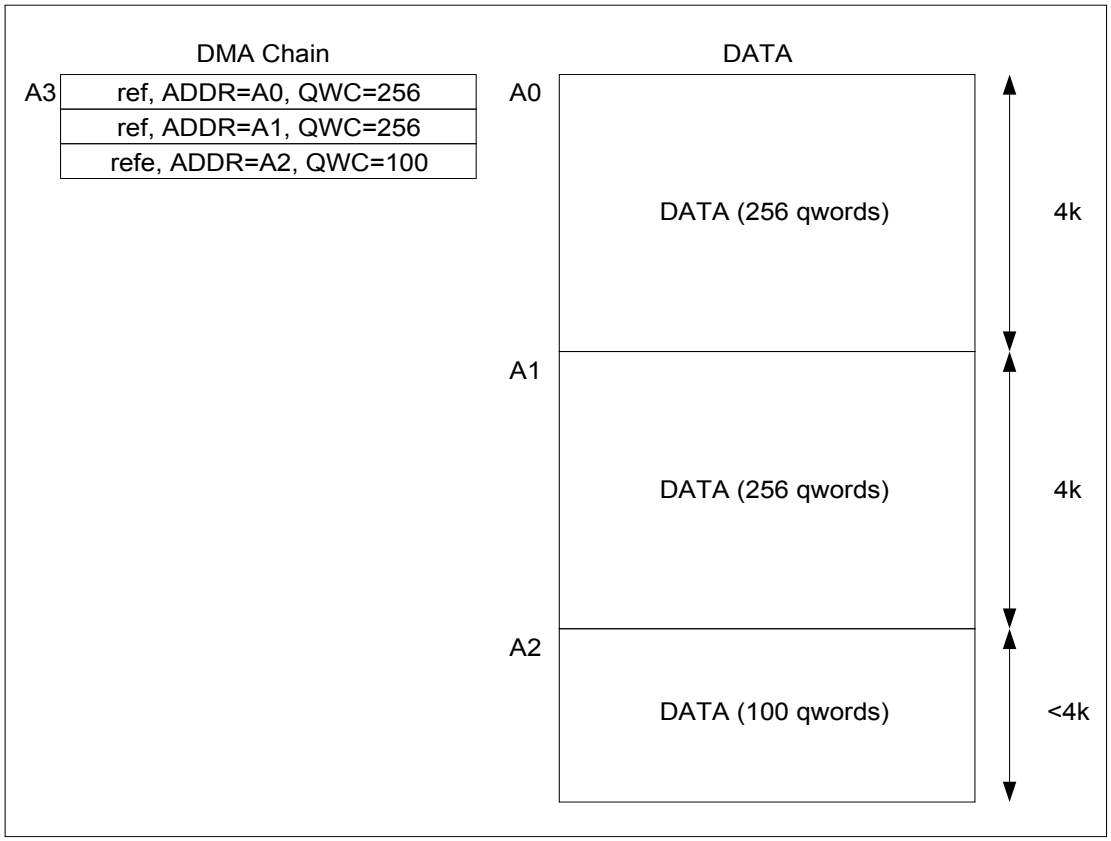


Figure 3

There are relative advantages to both of these methods of memory stitching. The use of **cnt** and next **tags** requires only one area of memory to be configured, whist the use of **ref** tags requires two areas of memory but only about half the number of tags.

Using Call Chains

Each of the DMAC channels to VIF0, VIF1 and the GIF contain tag address save registers which can be used to facilitate the creation of data subroutines. Data subroutines are similar to normal program subroutines in that once called, the subroutine performs it's function then returns control back to the main line of execution.

An example of a call chain is illustrated in figure 4. The data section at the right of the figure is stitched together into as large a packet as required and is ended with a return (**ret**) tag. The organisation of the data into this format would be undertaken prior to run time. The transfer is initiated when the DMAC reads the first **call** tag from the start of the call chain shown on the left hand side of figure 4.

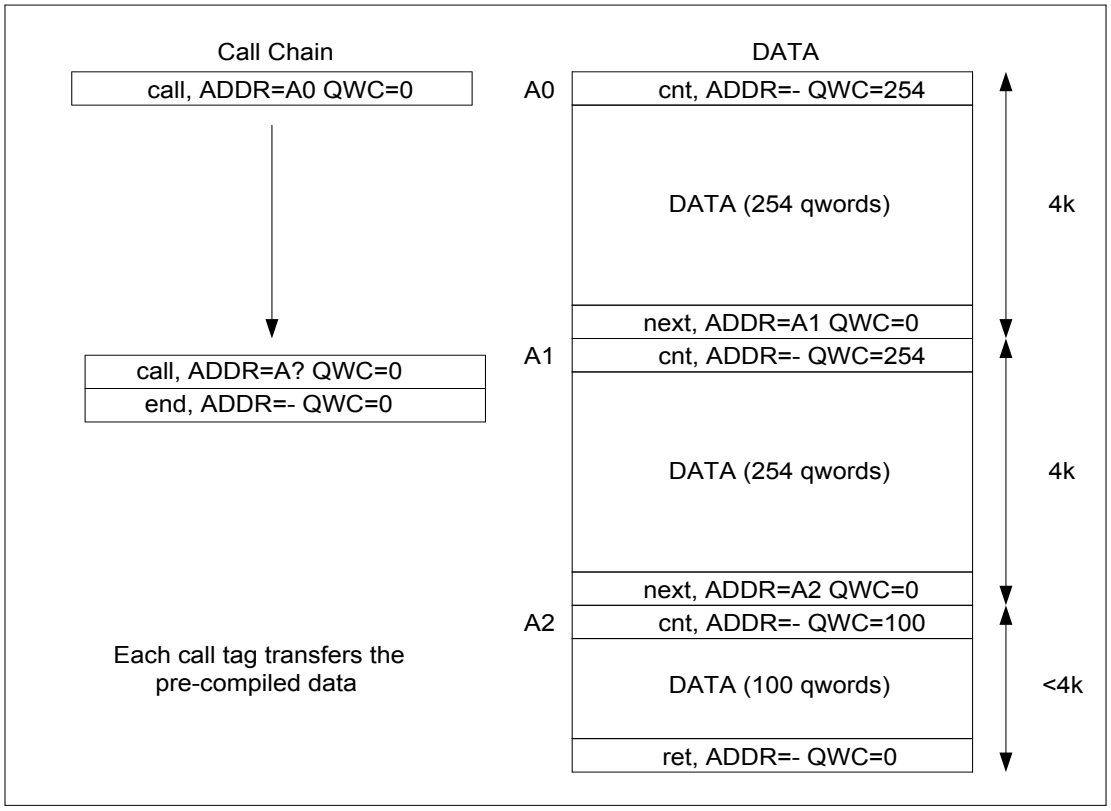


Figure 4

On reading the first **call** tag from the call chain, the DMAC pushes the following qword (which in this case is the next **call** tag) onto the call stack and reads the qword pointed to by the ADDR field in the call tag as the next tag. This action is carried out since the qword count (QWC) field of the **call** tag is set to zero. DMAC control then passes to the first **cnt** tag in the data section which is the first qword of the stitched data to be transferred. When the DMAC reads the **ret** tag at the end of the data, it transfers the number of qwords following this tag (which in this case is zero) then reads the qword popped from the call stack as the next tag. The next tag will thus be the second call tag in the call chain. This process repeats until the final **end** tag is reached in the call chain and the transfer is ended.

Using Call Chains in Games Programming

Now that the process of creating and transferring pre-compiled data chains has been describes, the use of such techniques in the writing of games programs will be discussed.

Consider the situation of a game consisting of several animated 3D models which must be sent down the graphics pipeline for rendering. It is advisable in such situations to cull as many objects as possible from the pipeline as early as possible within the pipeline thus saving valuable processing time. A simple, first approximation method might be to generate bounding spheres round each model and test each sphere against the view

frustum. Models inside or partly inside the frustum will require further processing whilst models fully outside the frustum can be culled. Consider therefore the pseudo-code shown in figure 5:

```
Main chain:
     Test visibility of model1;
     if visible (CALL pointing to Subchain1);
     Test visibility of model2;
     if visible (CALL pointing to Subchain2);
     END

Subchain1:
     REF pointing to model1 texture;
     REF pointing to model1 matrix data;
     REF pointing to model1 vertex data;
     RET

Subchain2:
REF pointing to model2 texture;
REF pointing to model2 matrix data;
REF pointing to model2 vertex data;
RET
```

Figure 5

In the main chain, the visibility of each model is checked and the appropriate sub chain is only called if the model is visible, thus requiring further processing.

Another use of call chains in games programming is in the rendering of animated models in either 2 or 3 dimensions. Consider that the data for an animated model is precompiled and organised in the manner shown in figure 6.

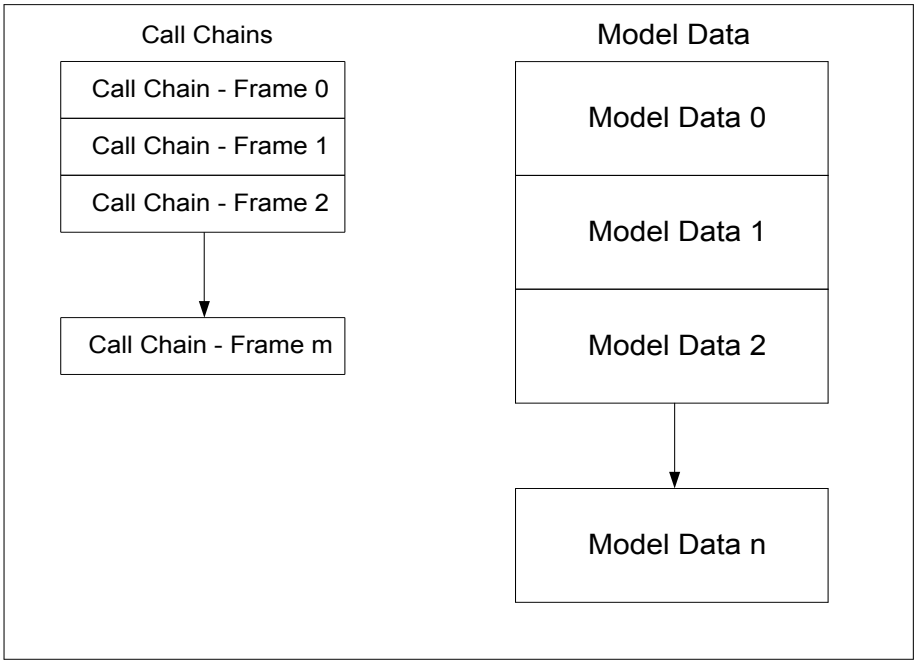


Figure 6

All of the data necessary to render any animation frame for the model is contained within the model data section. Various call chains are configured within the call chain section to call the appropriate model data needed for a specific animation frame. For example, the call chain for frame 0 may call the model data sections 0, 1, 2, 7, 9 and 12; the call chain for frame 6 may call the model data sections 0, 1, 5, 7, 10 and 11. Given that the data is pre-compiled into the correct format, it is thus possible to quickly render a specific animation frame for a model at run time with minimal processing overheads.

Conclusions

This article has illustrated the use of DMATags for the organisation of precompiled data within a computer game application. Pre-compiling and efficiently organising data prior to run time is essential in order to achieve effective application performance.

Acknowledgements

Much of the information presented here has been gleaned from various post on the Playstation2-linux.com developer forum. The author is grateful to the many contributors to this forum.

Dr Henry S Fortuna
Lecturer in Computer Games Technology
University of Abertay Dundee, Scotland UK
h.s.fortuna@abertay.ac.uk
27 April 2004

Features

Feature: MSKPATH3 Tutorial and Comment

Steven Osman (Sauce) sauce@users.playstation2-linux.com
Department of Computer Science
Carnegie Mellon University

This article is based on a an earlier post from Sauce at the end of March.
https://playstation2-linux.com/forum/message.php?msg_id=42234

First of all, we should all start by understanding that when using MSKPATH3, you're basically going to have **two** long chains. One chain will be sent through DMA channel 1 -- it's a VIF1 chain. The other chain will be sent through DMA channel 2 -- the GIF. The GIF channel will be used only to send textures, so basically the chain is a sequence of textures. Thorough VIF1, you'll be sending your geometry, matrices, whatever else you want, and a special sequence of MSKPATH3 VIF codes.

So in short, VIF1 gets geometry and MSKPATH3 instructions, while the GIF (through PATH 3) gets textures.

What MSKPATH3 is used for is to actually block the GIF from getting data through PATH 3. You can still send data to the GIF through PATH 1 & PATH 2 (i.e. through VU1 or VIF1), but PATH 3 will be blocked.

"First of all, we should all start by understanding that when using MSKPATH3, you're basically going to have **two** long chains."

So now we have a sequence of geometry going to VIF1, and a sequence of textures going to the GIF through PATH 3, and a tool that allows us to suspend/resume PATH 3 transfers.

Having said all that, let's consider the diagram that Hikey has posted [***Ed***. *See Hikey's comment at the end of this article for this diagram*]. You'll notice that he's using 3 sets of geometry with 3 different textures. It's important, of course, that the CURRENT texture is already loaded by the time the geometry goes to draw. It's also important that the CURRENT texture is not over-written by the NEXT texture while it is still in use by geometry. Finally, to get the benefit of MSKPATH3, the key is to have your NEXT texture(s) being uploaded while your CURRENT geometry is being used.

As soon as I mention "CURRENT" and "NEXT" you should be thinking to yourself "well, that sounds like double buffering to me!" Because it sure is! For simplicity, I'll assume we have two addresses to which we want to send textures, address 0 and address 128. Here's a simple timeline of what should happen:

0. Transfer texture 1 to address 0 through PATH 3 or 2
1. Wait for texture 1 to complete
2. Flush textures & activate texture 1
3. Start transfer of texture 2 to address 128 through PATH 3
4. Draw geometry using texture 1 at address 0
5. Wait for texture 2 to complete
6. Flush textures & activate texture 2
7. Start transfer of texture 3 to address 0 through PATH 3
8. Draw geometry using texture 2 at address 128
9. Wait for texture 3 to complete
10. Flush textures & activate texture 3
11. Draw geometry using texture 3 at address 0

If you skip step 0 as an initialization step (you could put that in PATH 2 for simplicity, for instance), you'll notice a pattern in steps 1-3 and steps 4-6. The pattern says "Wait for CURRENT texture in address A, start upload of the NEXT texture to address B, and draw using the CURRENT texture in address A). Of course, A and B swap every iteration as this is a double-buffering scheme.

Now remember that only the texture transfers are happening through PATH 3. Everything else, including the "Start transfer of texture..." are happening through VIF1. Some of the texture transfer & geometry drawing can happen at the same time, which is where the speed benefit comes from.

If you're still with me at this point, you have a conceptual understanding of how interleaving the texture transfer with the geometry drawing can be used to give a performance benefit. Now we should consider how MSKPATH3 works for a minute so we can get to the details that Sparky was trying to cover.

As I mentioned earlier, MSKPATH3 allows you to suspend & resume PATH 3 transfer. So trivially, whenever I say "Start transfer of texture n+1 ..." that simply means using MSKPATH3 to re-enable PATH 3. The problem is, when do we disable it again so that texture n+2, texture n+3, and so on don't also get uploaded? If we disable it too soon, would we disrupt the transfer of texture n+1, the texture we really wanted to transfer?

Well, we're fortunate in that when you use MSKPATH3, disabling PATH 3 doesn't abruptly terminate the PATH 3 transfer. Instead of saying, "Suspend PATH 3 transfers IMMEDIATELY," what MSKPATH3 is really saying is, "Suspend PATH 3 transfers AT THE END OF THIS CURRENT GS PACKET."

For a brief review, we see from page 150 of our EE User's Manual that a GIF primitive is a GIF code + data and a GS packet is a sequence of GIF primitives terminated by a GIF primitive that has EOP=1. In other words, a GS packet is any number of GIF codes + data with EOP=0, plus one GIF code + data at the end with EOP=1.

So the magic part here is that to get a texture to go through completely, we just need to make sure that all GIF primitives for a texture upload (except for the last one) have EOP=0, and the last one has EOP=1. When we'd like to "Start transfer of texture n+1" from VIF1, what we're basically adding to our sequence of VIF commands is:

a) MSKPATH3(0) to start the transfer
b) NOPs in order to wait long enough to ensure that the transfer has started
c) MSKPATH3(0x8000) to suspend PATH 3 at the end of the current transfer

The only particularly tricky part is answering "how many NOPs do we put in?" If you put in too many NOPs, there's the possibility that if texture n+1 is very small, it'll get transferred immediately and texture n+2 will also get started. If you put in too few NOPs, there's the possibility that texture n+1 will never even get started because the GIF won't have had a chance to even start transferring that first GIF code with EOP=0 indicating the beginning of the GS packet.
Understanding that too many NOPs could make two transfers go through instead of one is fairly simple, but you may wonder, "why would it take so long for a transfer to start? Why would I ever suffer the too few NOPs problem?" There are a couple of reasons for that.

First, remember that only one thing can read from the EE memory at once. This means that if you've got a number (even just the VIF1 and GIF) transfers from the EE memory going and using the EE memory in your own program, it could take a little bit of time.

Second, remember that if you use many CALL and NEXT DMA tags in your GIF transfer, the DMAC may take a little time processing and following all those DMA tags that it never gets to transfer that first GIF code.

The magic then is to try to put the GIF primitive with EOP=1 from texture n as close as possible to a GIF code with EOP=0 for texture n+1. This means that once texture n ends, the very next thing to be transferred by the DMAC is the first GIF code of the next texture -- not a bunch of CALL tags or anything else.

Sparky proposed a pretty good solution to that in his post, I'll give a slightly different example that may waste a tiny bit of memory more but is simpler to understand. Let's say we want to pre-build our textures so that they can either upload to address 0, or address 128. We'd create sequences in memory as in Snippet B.1.

```
tex1_addr0:
 DMA_NEXT(ADDR=tex1_data, QWC=5)
 giftag(eop=0, nloop=4, reg=a+d)
 BITBLTBUF(upload address addr=0)
 TRXDIR
 TRXPOS
 TRXREG

tex1_addr128:
 DMA_NEXT(ADDR=tex1_data, QWC=5)
 giftag(eop=0, nloop=4, reg=a+d)
 BITBLTBUF(upload address addr=128)
 TRXDIR
 TRXPOS
 TRXREG

tex1_data:
 DMA_NEXT Tags (as needed to stitch)
 giftag(eop=0, nloop=data size)
 texture data
 DMA_RET
```

Snippet B.1.

Note first that I didn't have anything in there with EOP=1. You'd create one of these sets for tex1, one for tex2, and so on. Now as you're creating your PATH 3 chain to upload these textures (let's assume that we start with texture 2 at address 128 since we said that texture 1 may have been

uploaded through PATH 2 or some other method as part of initialization), your sequence would look like this:

```
render_loop_path3_upload_chain:
 DMA NEXT(ADDR=uploadtex2, QWC=1)
 giftag(eop=0, nloop=0, nreg=0)

uploadtex2:
 DMA CALL tex2_addr_128
 DMA NEXT(ADDR=uploadtex3, QWC=2)
 giftag(eop=1, nloop=0, nreg=0)
 giftag(eop=0, nloop=0, nreg=0)

uploadtex3:
 DMA CALL tex3_addr_0
 DMA NEXT(ADDR=uploadtex4, QWC=2)
 giftag(eop=1, nloop=0, nreg=0)
 giftag(eop=0, nloop=0, nreg=0)

uploadtex4:
 [...]
```

Snippet B.2.

Of course, make sure you're stitching this chain as needed. What has this achieved?

Well, if you look closely you'll see that the transfer IMMEDIATELY begins with a GS packet. Similarly, every time a GS packet ends (which is the GIFtag with EOP=1), the next GS packet begins immediately after it. This helps to give us a much more predictable (and short) amount of time that will be required to start the next GS packet when MSKPATH3 temporarily re-enables PATH 3 transfers, helping to avoid the problem of the transfer getting missed because there aren't enough NOPs in the VIF1 sequence.

Two other details before I end this discussion...

1. The magic number of NOPs to put in your VIF1 sequence that appears to work well is 24. Don't ask, just learn it, love it, and use it. Keep in mind that really, really small textures may transfer faster than that, but it's not safe to reduce the 24 NOPs because you risk missing a texture entirely. The solution is to pad really small textures with other stuff (for instance, you could always add a bunch of GIF NOPs at the end of your texture).
2. When I mentioned "wait for texture n+1 to complete," the VIF tag you want to use is FLUSHA. FLUSHA makes sure that the path 3 transfer (which is texture n+1) has finished. Don't forget to put the appropriate instructions to activate the texture.

If anyone has any questions about this, please let me know. If any pros out there want to jump down my throat for any gross errors I've made, please do so, but do it gently :)

Sauce

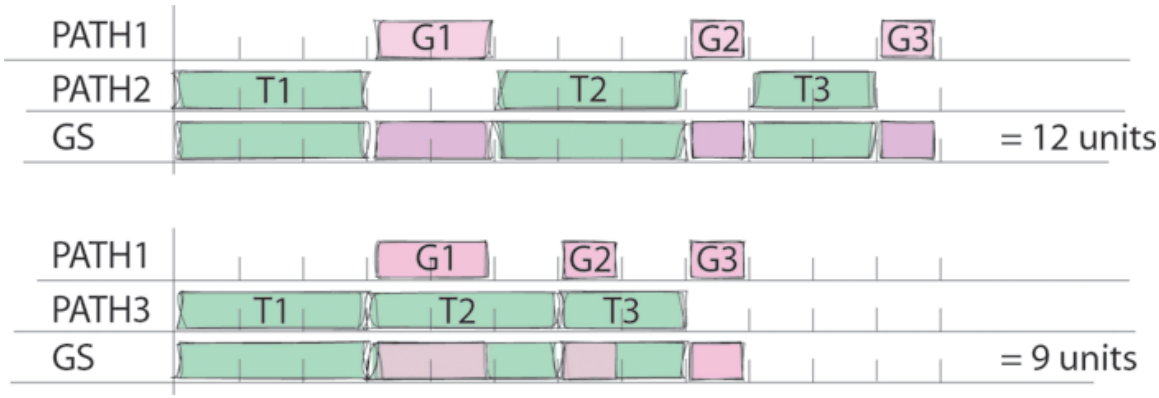
I would like to add a comment on the "typical" chain I gave in the mentioned post. This was if you use only PATH1/2, which is the easiest way to go. When this is sorted out, the next step is to use PATH1/3 instead. The reason for this is that using PATH1 and 2, the data goes through the VIF (typically) for both PATHs. So while you are uploading a texture, you cannot draw anything on the screen. If you have many textures, you might be slowing down the rendering unnecessarily.

The PATH1/3 technique is a lot more complicated. But it means that only the 3D geometry is going through the VIF and that in the gaps between rendering the polys, you could be sending the next texture to use.

This means you need:

- a double buffer in the VRAM (in your case 128k each), one for the texture that is currently used and one for the next texture.
- a better VRAM manager that can send the texture *before* you need it, otherwise you will still be waiting for it and this is all pointless. It needs to group the textures into batches too, it's useless to send only a tiny texture.
- a geometry/texture synchronisation technique. There are two main ones: Interrupts (which at the moment are not available through sps2) and MSKPATH3, which Sparky has implemented and is provided in the sps2 samples.

I knocked up a programmer's art diagram to illustrate:



http://www.scee.sony.co.uk/lionel/cfyc/path123.gif

As you can see, already with three objects the speed difference is significant.

Hikey (Lionel Lemarie, SCEE)

Interview

This month's interview is with the creators of the recently-released VUC C-like compiler for the Vector Processing Units on the Playstation 2. You may contact them through *peterp@users.playstation2-linux.com*

PSynergy: Where do you live?

We both live in Gothenburg, Sweden.

PSynergy: How old are you?

We're both 28 years old.

PSynergy: What do you do for a living?

Ola: I'm a student at the University of Gothenburg, currently working on my masters thesis, previously I've worked as a games programmer in Brisbane, Australia.
Peter: I've been working as a games programmer since 1995, never on a PS2 game though.

PSynergy: When did you get started on developing under PS2 Linux?

Ola: In fact, I haven't :) I did some PS2 while working professionally, but I haven't got a Linux kit (or indeed a PS2 even), we're doing all testing on Peters kit.
Peter: I've always wanted to get into console programming. Then I heard about the ps2linux kit about a year ago and ordered it. When I got it and started to read the manuals I got a bit hesitant, so I used ps2gl the first six months or so before I started to program it on a lower level.

PSynergy: What do you like and dislike about the PS2 hardware?

Ola: It's a fun piece of hardware with several processors and all, very versatile. On the downside is the fact that they completely crippled vu0 by giving it only 4 kb of memory.
Peter: Obviously I'm a big fan of the VU units. Comparing them to vertex shaders on a PC, it's really nice to be able to actually generate geometry on the fly and to have branching. Another thing I really like is that you can access the frame buffer in a more direct way than you normally can on a PC.

I'm not particularly impressed with the pixel pipeline. A few extra texture units and an extra VU unit for pixel processing would have been nice. And of course, you can never have enough memory.

PSynergy: How is your development environment set up, both physically and on your machine?

Peter: I got a small network set up with one Windows PC and one PS2linux box (using samba). I do all the source code editing on my PC running Visual Studio NET 2003. Telnet is used to compile and run the programs. In fact, it's just like Henry described in the "PlayStation2 in Higher Education" article in the February issue.
Ola: I run Visual Studio too, and a CVS server (since I've got a fixed IP), Then since I don't have a PS2...

PSynergy: What do you foresee as the future for the PS2 and Linux development?

Ola: Don't really have a great deal of a clue about the future, but it seems clear to me that the PS3 isn't gonna be out for another 2 years or so, and that's not the immediate end for PS2 either.
Peter: I read in a recent magazine that Sony said that even though they sold a lot of units, they figure that they've sold less than half of what they expect to sell during its lifetime. I guess that means that when the PS3 is released, we're going to see the same thing that happened to the PS1 when the PS2 was released. All the early adopters buy the latest and greatest and then give away their PS2 to some younger relative. And the PS2 will get smaller and so cheap that any impulse buyer not normally buying games might pick one up. The gaming population will grow and the PS2 games are going to be more mainstream and probably targeted to a younger audience.

As for the future of the ps2linux environment; I think and hope that it will get easier to develop for the kit as we get more and better documentation (Henrys sps2- and Eratos' VU tutorials, PSynergy etc), libraries and tools (e.g. Sauces' SPS2 & Visual VU Debugger).

Maybe the next step is to build more high-level libraries or rather components for people to pick from when putting together their own game (bjt's kiss renderer, sparky's intmd, etc).

PSynergy: Anything else you would like to add?

Ola: Don't you have any questions about VUC at all? ;)

PSynergy: Yes – maybe next time! Thank you for taking the time to respond to these questions.

PS2 Development Tips

Tip: Triangle Strip Stitching (posts from jawadx, sparky, sauce and hikey)

To draw a mesh or object with multiple vertices, it is useful to use Triangle Strips, specifically because they can be stitched together to make polygons. But doing so successfully requires correct usage of the PRIM tag and ADC bits. (See GS User's Guide). Incorrect usage results in broken strips, as brought up by jawadx in a post. You don't need to add extra vertices or anything like that, but the first two vertices of every strip should be delivered with **ADC** enabled which means drawing kick disabled.

Now we only have the problem limited to from one chunk to the next (between deliveries/**mscalls**). If, when you kick your vertices, you don't set the PRIM register in your GIFtag then it will be the same PRIM as it was for the previous kick and the vertex queue as it's called will remain **intact**. Of course you'll have to set the PRIM register once, perhaps at the start of a mesh.

This means you will not obtain split strips and you will NOT have to add any extra dummy vertices or anything like that. You can just process the vertices.

So, if you **never** send the PRIM register along with your gif packet that contains vertices, you should set ADC on the first two vertices of a strip. This is so when you move from one strip on to a second strip, it won't draw a triangle that spans the last one or two vertices of the first strip and the first one or two of the second strip.

If, on the other hand, you send PRIM whenever you're starting a new strip, then setting the ADC bit is optional because nothing would get drawn until the third vertex anyway.

So to make it 100% clear you always set the ADC bits at the 2 first vertices of every actual strip, if the next batch takes over a strip from the previous batch and it's processing somewhere after the first two vertices of this strip then it would not set the ADCs until the next strip.

Next Month in PSynergy

Look for PSynergy Next Month!